SOLVING PROBLEMS BY SEARCHING: INFORMED SEARCH

Russel, Norvig, "Artificial Intelligence, A Modern Approach"

Lecture outline

- Informed search strategies
- Best-first searches
  - Greedy best first
  - A*
- Heuristics
- Local search strategies
  - Hill climbing
  - Simulated annealing
  - Local beam search

Best-first search

Informed search algorithms use problem-specific knowledge to find solutions more efficiently

- h(n), heuristic function

Best-first searching algorithm

- uses evaluation function f(n) (measuring distance to the goal)
- the node with lowest value for f(n) is expanded next.
- implements the fringe as a queue with ascending values for f(n)

Several best-first algorithms depending on the evaluation function

- greedy best-first search
- A* search

A heuristic function

There is a whole family of BFS algorithms using different evaluation functions.

A key component of these algorithms is a heuristic function, denoted $h(n)$:

> [dictionary] *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*

- A heuristic function takes a node as input but depends only on the state at that node.
- $h(n)$ = estimated cost of the cheapest path from node *n* to goal node.
- If *n* is goal then $h(n)=0$

Heuristic functions are the most common form in which additional knowledge of a problem is imparted to the search algorithm.

Greedy best-first search

Evaluation function

- $f(n) = h(n)$ (= estimate of the lowest cost from *n* to *goal*)
- e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest

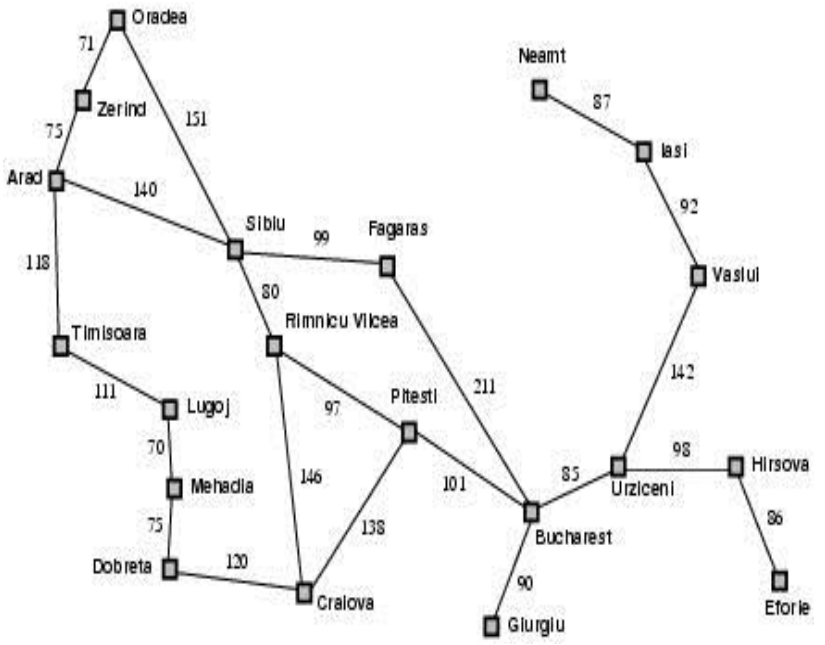
Greedy best-first search expands the node that ***appears*** to be closest to goal

Greedy best-first search

Lets apply GBFS to the Romanian route finding problem using the straight-line distance heuristic (h_{SLD})

Assuming Bucharest is the goal, we need to know the straight line

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



So: $h_{SLD}(In(Arad)) = 366$

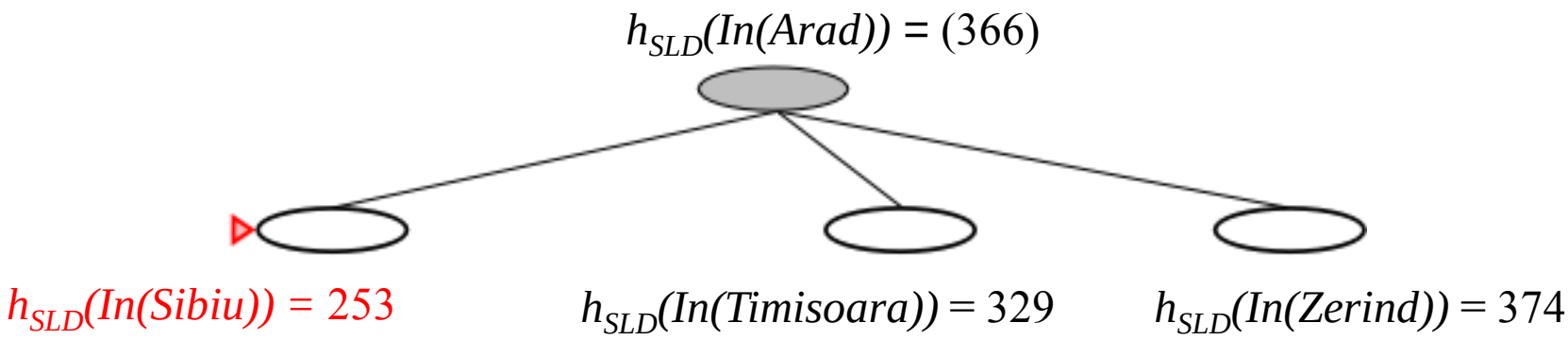
Note: these distances cannot be computed from the problem description itself!

Greedy best-first search

$h_{SLD}(In(Arad)) = (366)$

The initial state=Arad

Greedy best-first search

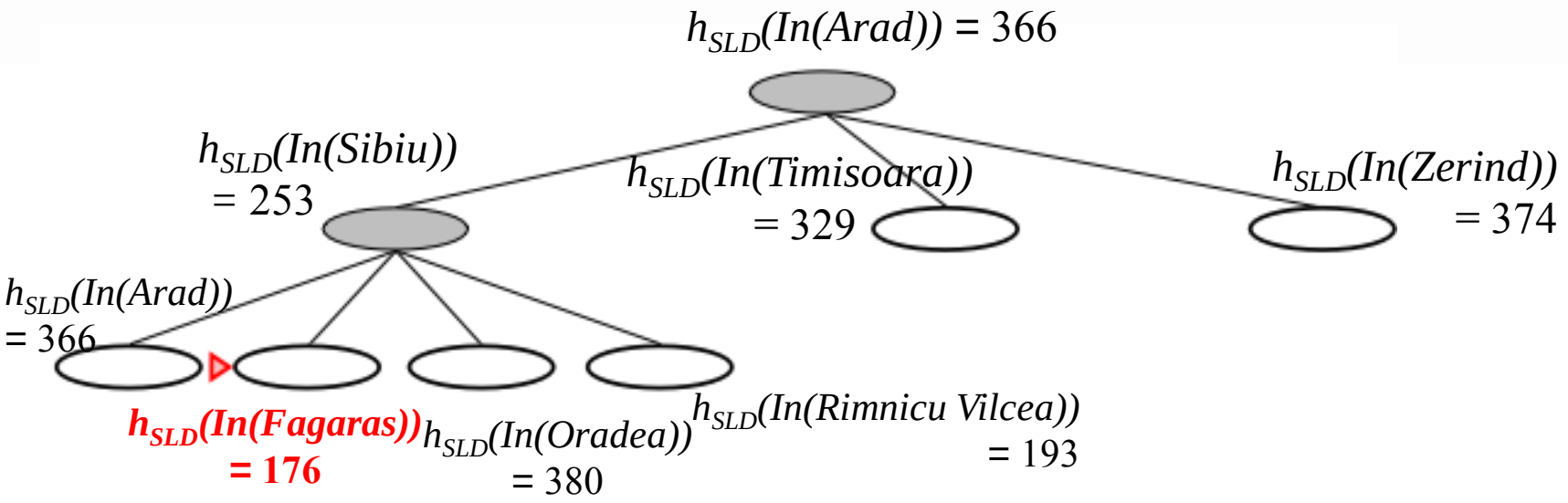


The first expansion step produces:

- Sibiu, Timisoara and Zerind

Greedy best-first will select Sibiu because it is closer to Bucharest than the other alternatives.

Greedy best-first search

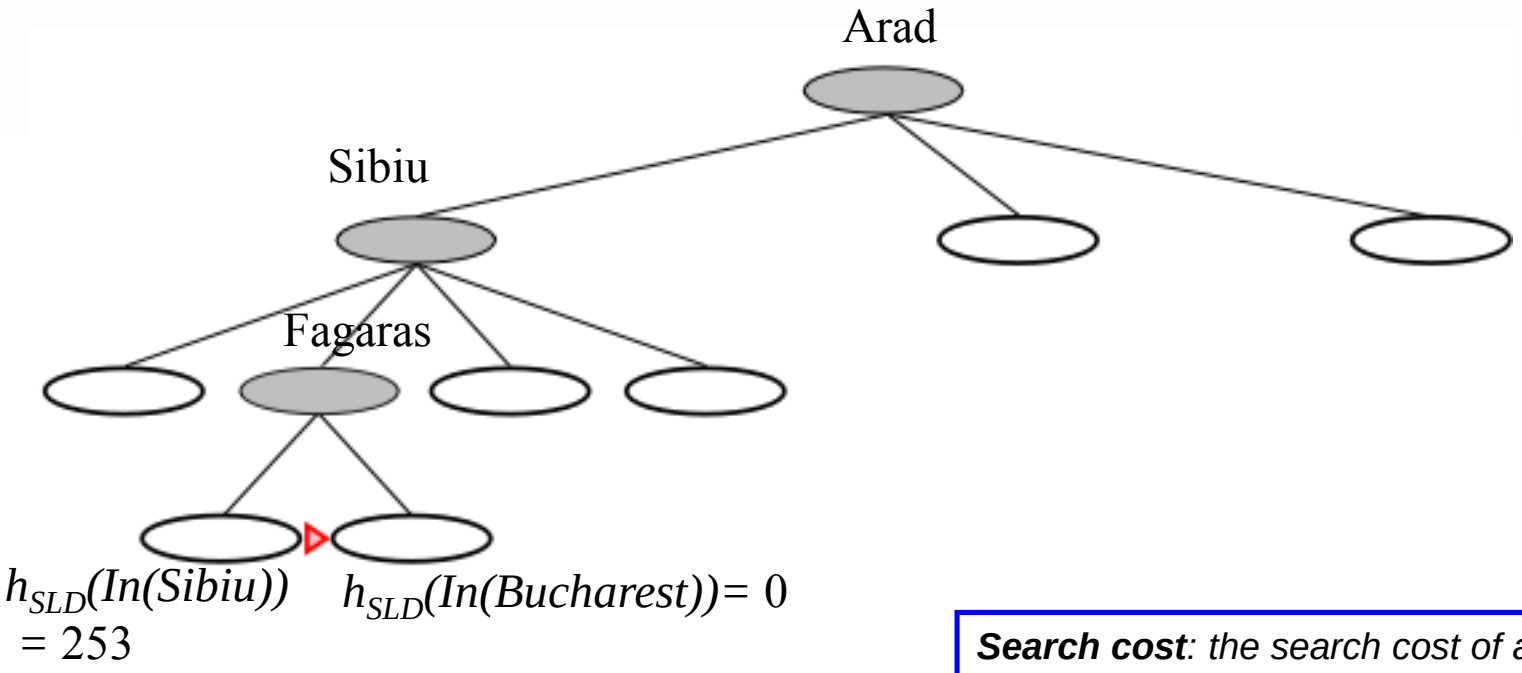


If Sibiu is expanded we get:

- Arad, Fagaras, Oradea and Rimnicu Vilcea

Greedy best-first search will select: Fagaras

Greedy best-first search



***Search cost**: the search cost of an algorithm typically depends on the time complexity but can also include a term for memory usage.*

If Fagaras is expanded we get:

- Sibiu and Bucharest

Goal reached!

- For this particular problem, GBFS using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost in minimal!
- Yet not optimal, the path via Sibiu and Fagaras to Bucharest is 32 km longer than the path [Arad, Sibiu, Rimnicu Vilcea, Pitesti, Bucharest]

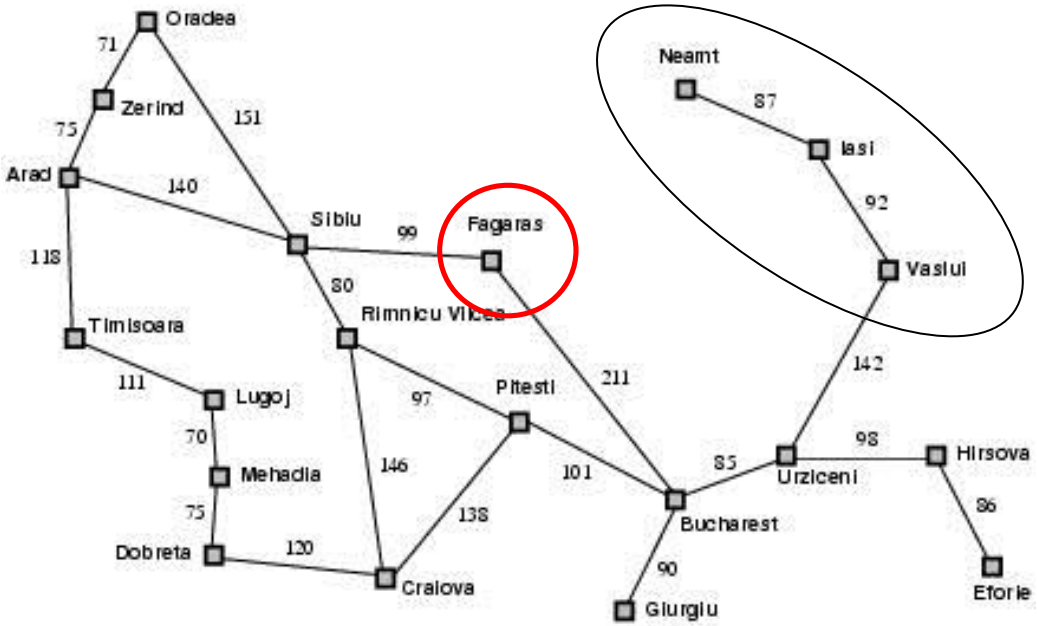
Properties of Greedy BFS

Minimising h(n) can result in false starts:

- Consider the problem of getting from Isasi to Fagaras. h_{SLD} suggests that Neamt be expanded first because it is closest to the goal, but it is a dead end! In this case, the h_{SLD} causes unnecessary nodes to be expanded.

Complete - not

- If we are not careful to detect repeated states the solutions will never be found, the search will oscillate between Neamt and Iasi for a goal Fagaras(similar to DFS).



Properties of Greedy BFS

Greedy BFS resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead-end.

It suffers from the same defects as DFS: not optimal and incomplete.

Time complexity

- $O(b^m)$, where m is the maximum depth of the search space (same as worst case DFS). **However, with a good heuristic the complexity can be dramatically improved**

Space complexity

- $O(b^m)$ keeps all nodes in memory.

A* search

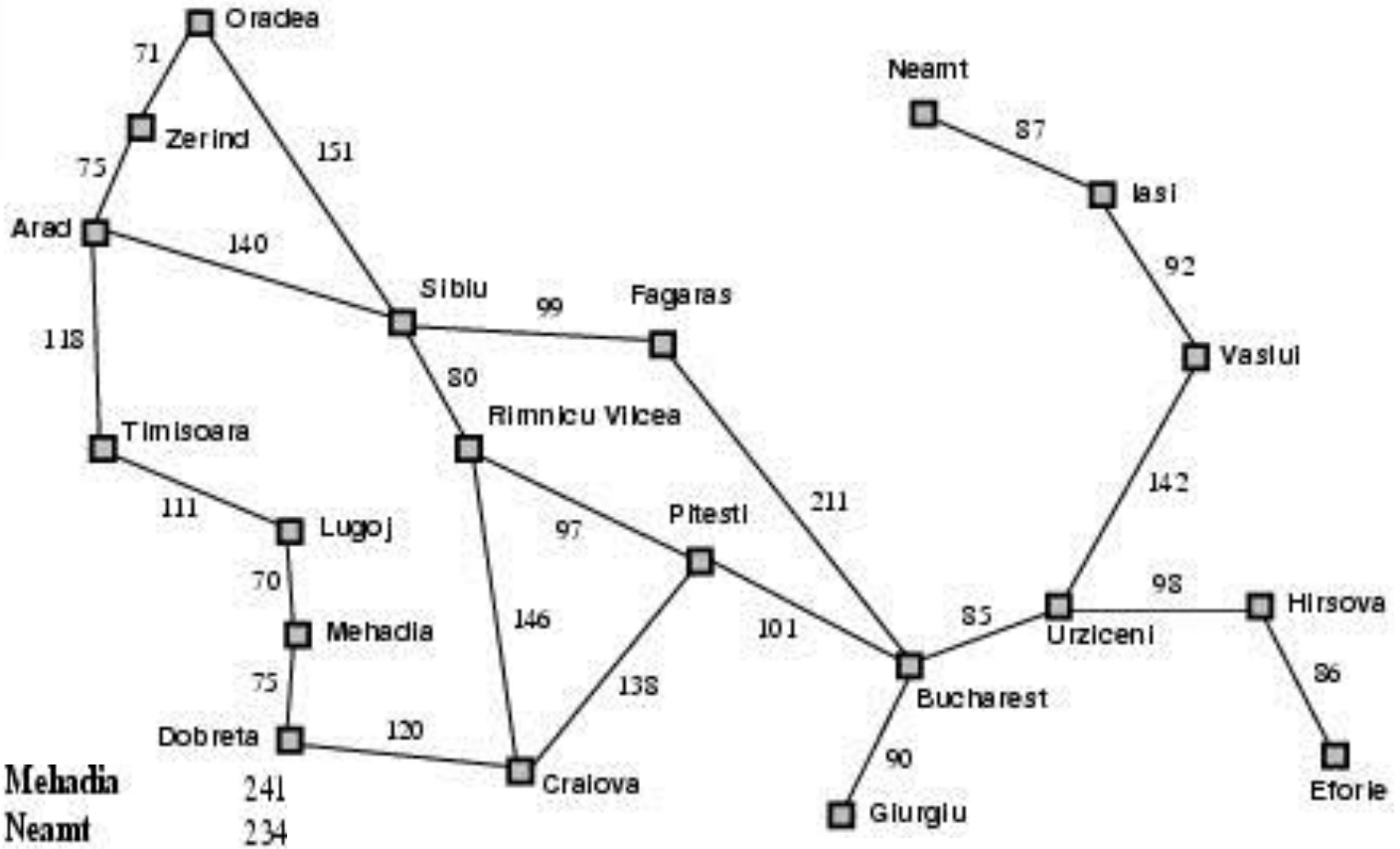
Best-known algorithm in the Best-First Search family.

Evaluation function

$f(n)=g(n) + h(n)$

- $g(n)$ the cost (so far) to reach the node.
- $h(n)$ estimated cost to get from the node to the goal.
- $f(n)$ estimated total cost of path through n to goal.

Romania example

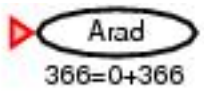


Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244

Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

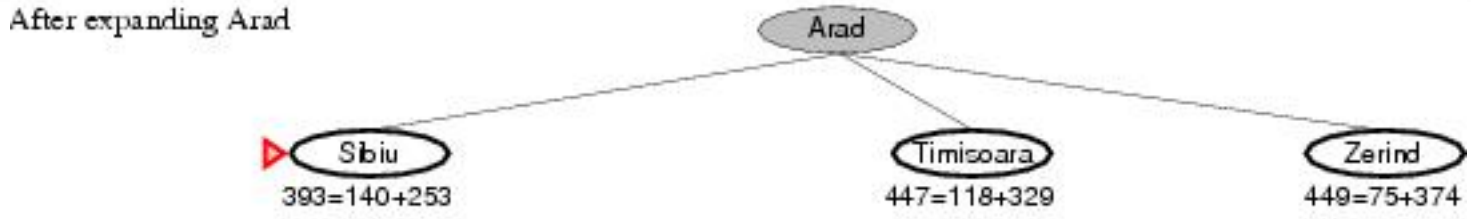
(a) The initial state



Find Bucharest starting at Arad

- f(Arad) = g(Arad,Arad)+h(Arad)=0+366=366

A* search example

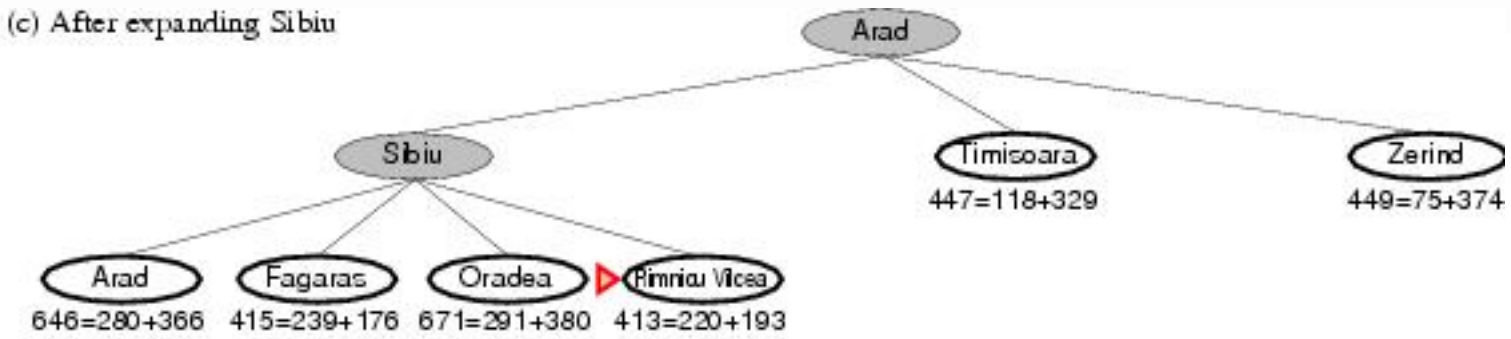


Expand Arad and determine *f(n)* for each node

- f(Sibiu)=g(Arad,Sibiu)+h(Sibiu)=140+253= **393**
- f(Timisoara)=g(Arad,Timisoara)+h(Timisoara)=118+329= **447**
- f(Zerind)=g(Arad,Zerind)+h(Zerind)=75+374= **449**

Best choice is Sibiu

A* search example

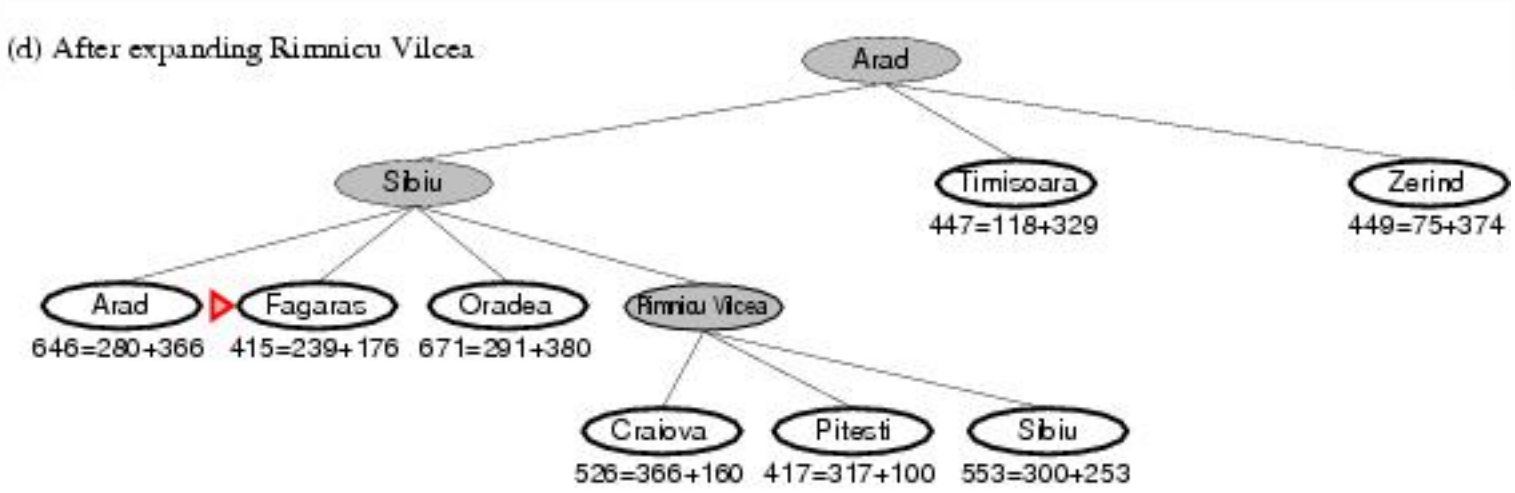


Expand Sibiu and determine *f(n)* for each node

- f(Arad)=g(Sibiu,Arad)+h(Arad)=280+366=646
- f(Fagaras)=g(Sibiu,Fagaras)+h(Fagaras)=239+179=415
- f(Oradea)=g(Sibiu,Oradea)+h(Oradea)=291+380=671
- f(Rimnicu Vilcea)=g(Sibiu,Rimnicu Vilcea)+
  h(Rimnicu Vilcea)=220+192=413

Best choice is Rimnicu Vilcea

A* search example

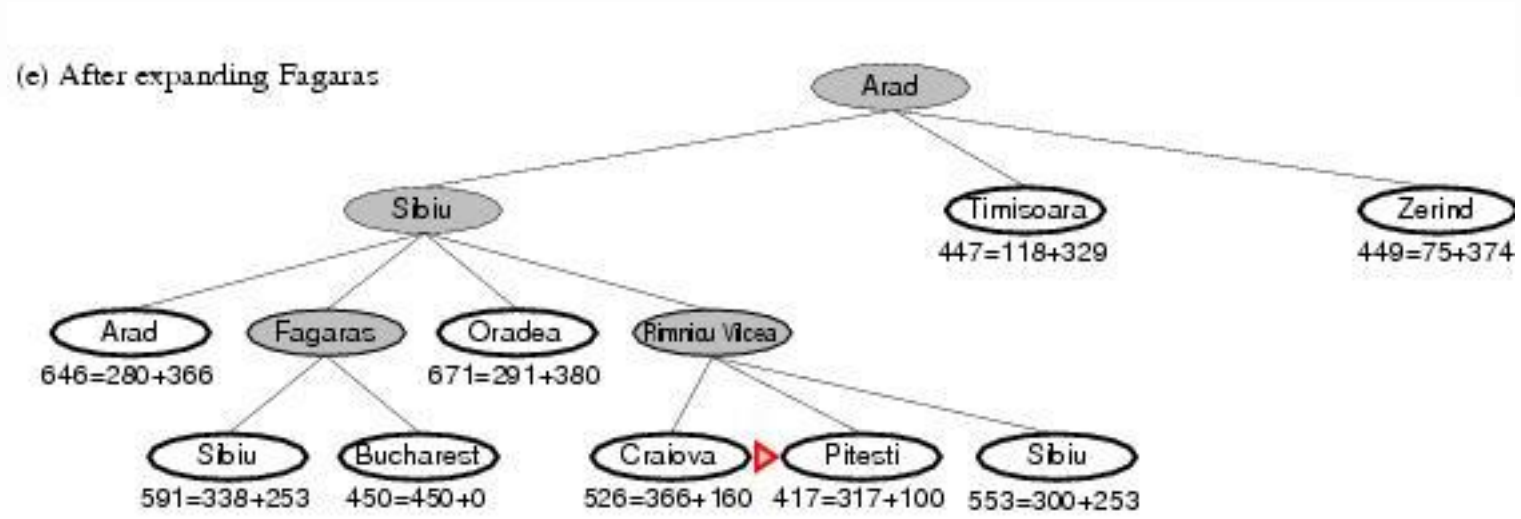


Expand Rimnicu Vilcea and determine *f(n)* for each node

- f(Craiova)=g(Rimnicu Vilcea,Craiova)+h(Craiova)=360+160=526
- f(Pitesti)=g(Rimnicu Vilcea, Pitesti)+h(Pitesti)=317+100=417
- f(Sibiu)=g(Rimnicu Vilcea,Sibiu)+h(Sibiu)=300+253=553

Best choice is Fagaras

A* search example



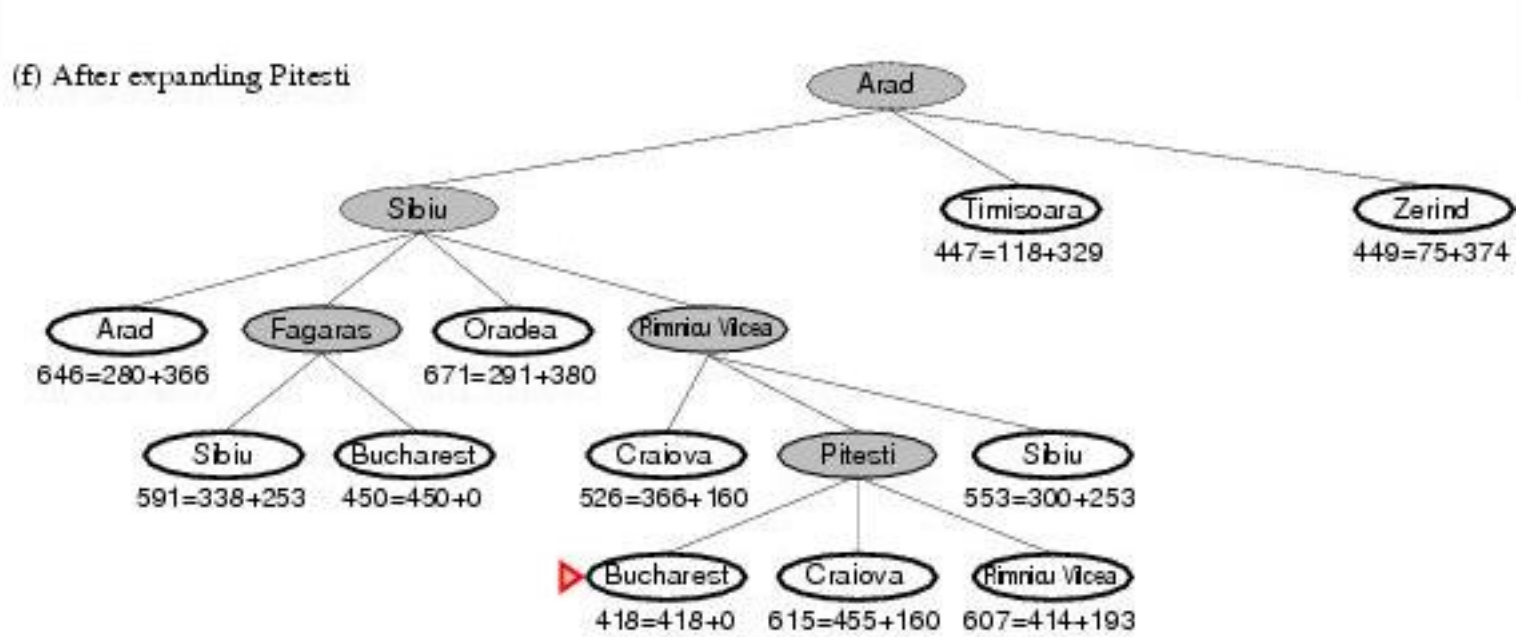
Expand Fagaras and determine *f(n)* for each node

- f(Sibiu)=g(Fagaras, Sibiu)+h(Sibiu)=338+253=591
- *f(Bucharest)=g(Fagaras,Bucharest)+h(Bucharest)=450+0=450*

Best choice is Pitesti

- Bucharest is not selected for expansion because its f-cost (450) is higher than that of Pitesti's (417). Another way to view this is that there might be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that cost 450.

A* search example



Expand Pitesti and determine *f(n)* for each node

- *f(Bucharest)=g(Pitesti,Bucharest)+h(Bucharest)=418+0=418*

Best choice is Bucharest

- Optimal solution (only if *h(n)* is admissible)

A* search

A heuristic is **admissible** if it *never overestimates* the cost to reach the goal.

Admissible Heuristic

Admissible heuristic functions are naturally optimistic because they think the cost of solving a problem is less than it actually is.

Formally:

1. $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n
2. $h(n) >= 0$ so $h(G)=0$ for any goal G.

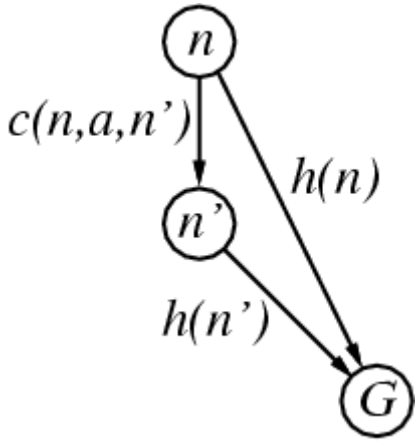
E.g. $h_{SLD}(n)$ never overestimates the actual road distance

Consistent heuristics

A heuristic is **consistent (monotonic)** if for every node *n*, every successor *n'* of *n* generated by any action *a*,

$$h(n) \leq c(n,a,n') + h(n')$$

where *c(n,a,n') is the step cost for getting from n to n'*



If *h* is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

i.e., **f(n') ≥ f(n)** , or

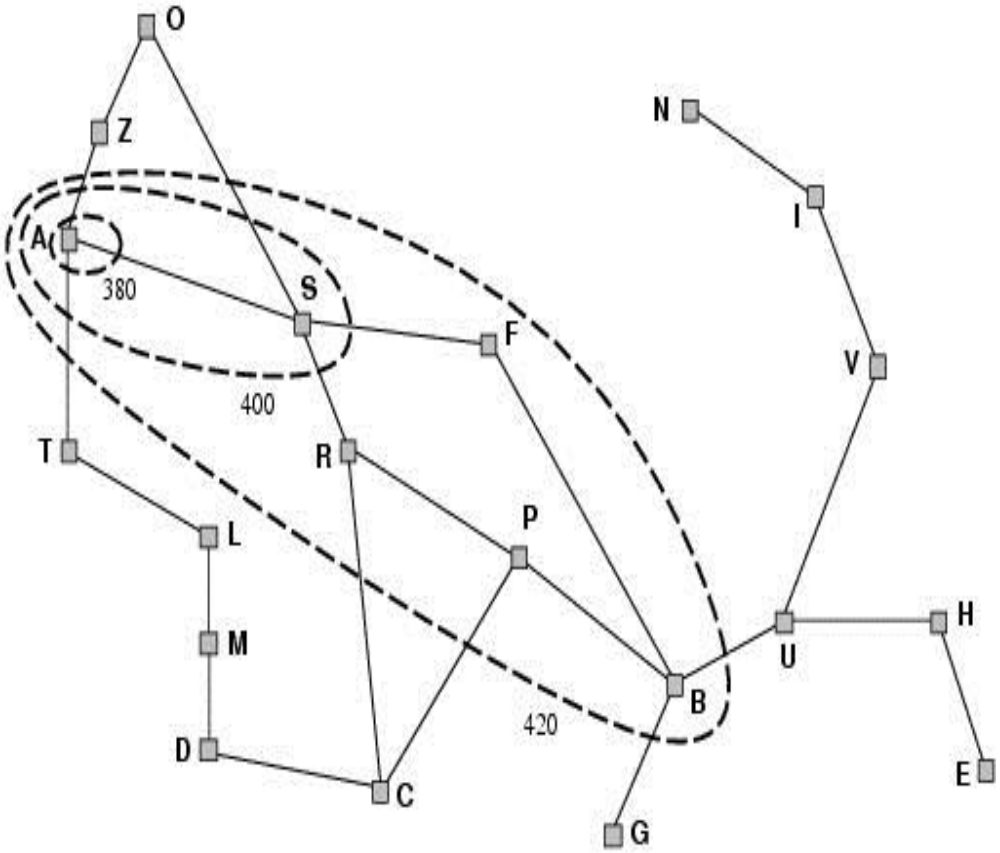
f(n) is non-decreasing along any path.

Optimality of A*

The fact that f-costs are non-decreasing along any path also means that we can draw contours in the state space, just like the contours in a topographic map.

Inside the contour labeled 400, all nodes have f(n) ≤ 400, and so on.

Then, because A* expands the fringe node of lowest f-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.



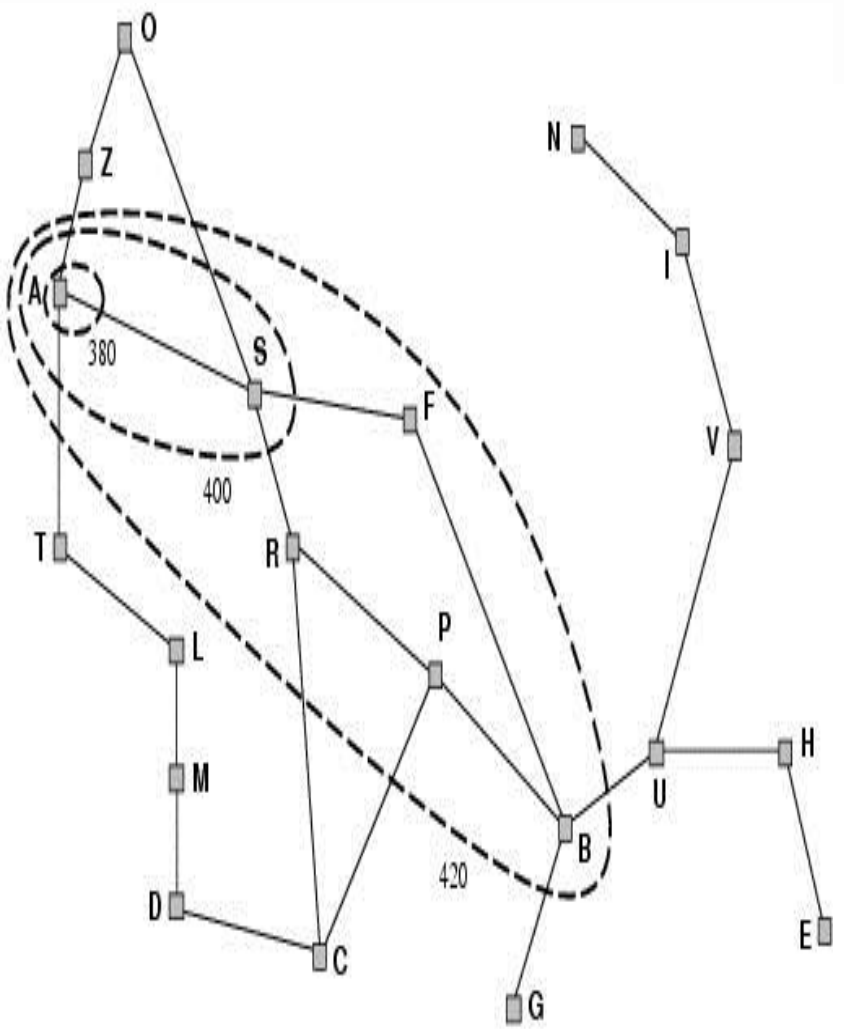
Optimality of A*

With uniform cost search (A* search using h(n) =0) the bands will be "circular" around the start state.

With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.

If C* is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with f(n) < C*
- A* might then expand some of the nodes right on the "goal contour" (where f(n) = C*) before selecting a goal node.

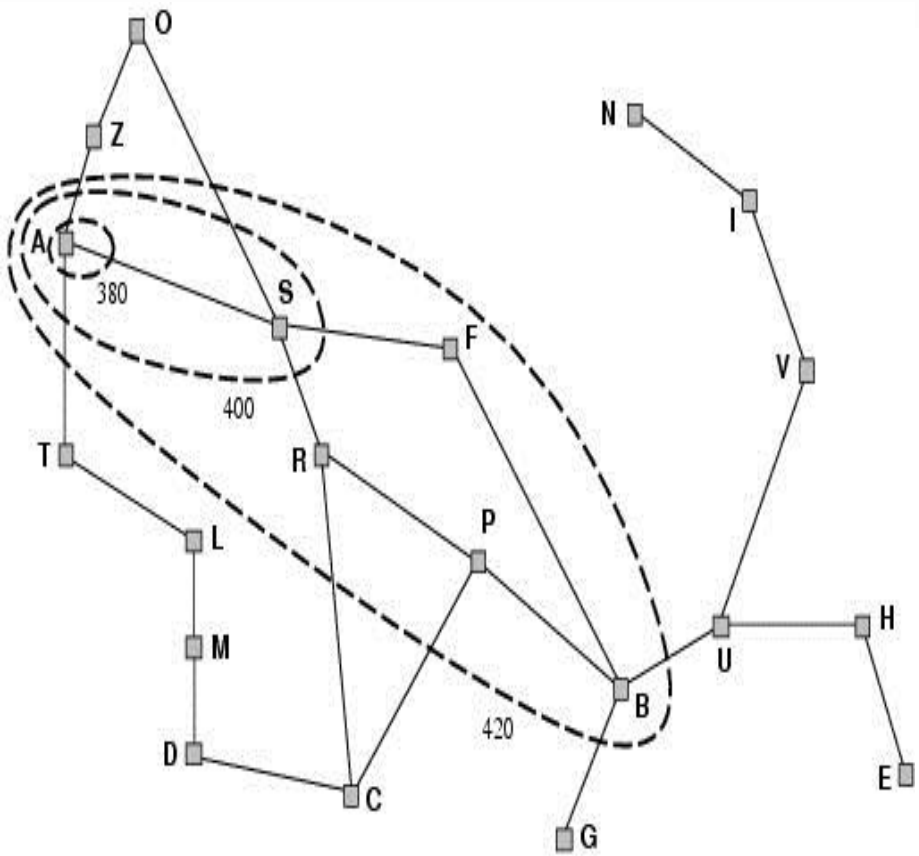


Optimality of A*

Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have a higher f-cost, and thus higher g-cost (because all goal nodes have h(n) =0).

Intuitively, it is also obvious that A* search is complete.

- As we add bands of increasing f, we must eventually reach a band where f is equal to the cost of the path to a goal state.



Properties of A* search

Completeness

- Since bands of increasing *f* are added
- Unless there are infinitely many nodes with $f<f(G)$

Optimal

- Cannot expand f_{i+1} until f_i is finished.
- A* expands all nodes with $f(n)< C^*$
- A* expands some nodes with $f(n)=C^*$
- A* expands no nodes with $f(n)>C^*$

Time complexity X

- For most problems the number of nodes within the goal contour search space is still **exponential** in the length of the solution.

Space complexity X

- Keeps all generated nodes in memory
- Hence space is the major problem not time

Dominance

If $h2(n) \geq h1(n)$ for all n (both admissible)

then h2 dominates h1

h2 is better for search

A* using h2 will never expand more nodes than A* using h1

- All nodes n with $f(n)<C^*$ (cost of optimal solution) will be expanded, or in other words

  all nodes n with $h(n)<C^* - g(n)$ will be expanded

- Since $h2(n) \geq h1(n)$ all nodes expanded with h2 will also be expanded with h1 and h1 may cause more nodes to be expanded

Dominance

Sample search costs (average number of nodes expanded) for set of 1200 problems with solution length 2 to 24 solved used Iterative deepening search, A* using h1 and A* using h2

- d=12 IDS = 3,644,035 nodes
  A*(h1) = 227 nodes
  A*(h2) = 73 nodes

- d=24 IDS = too many nodes
  A*(h1) = 39,135 nodes
  A*(h2) = 1,641 nodes

Inventing admissible heuristics

A problem with fewer restrictions on the actions is called a **relaxed problem**

Relaxing the 8-puzzle:

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to **any adjacent square** (even occupied), then $h_2(n)$ gives the shortest solution

The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem

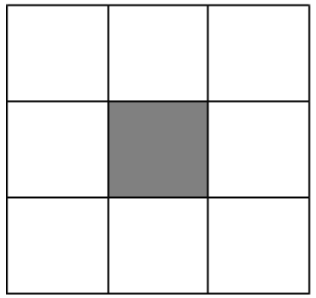
Inventing admissible heuristics

If a problem definition is written in a formal language, it is possible to construct relaxed problems automatically

The relaxed problems should be solved without search

For example:

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B **AND** B is blank



will generate

(1) A tile can move from square A to square B if
A is horizontally or vertically adjacent to B ← h2

(2) A tile can move from square A to square B if
B is blank

(3) A tile can move from square A to square B ← h1

Part 1: Summary

Best First Search

- Priority Queue ordered by f(n)
- Heuristics and Admissibility
- Greedy Best First Search f(n) = h(n)
- A* Best First Search f(n) = h(n) + g(n)

Local search and optimization

The search algorithms that we have examined so far are designed to explore search spaces **systematically**.

- This systematic search is achieved by keeping one or more paths in memory and by recording which alternatives have been explored along the path and which have not.
- When a goal is found, the *path* to that goal constitutes a solution to the problem.

In many problems, however, the path to the goal is irrelevant; e.g., 8 queens, integrated-circuit design, factory-floor layout, job-shop scheduling ...

If the path to the goal does not matter we can use a different set of algorithms that do not worry about paths at all.

Local search and optimization

Local search algorithms operate using a single current state (rather than multiple paths) and generally move only to neighbours of that state.

Typically, the paths followed by the search are not retained.

Advantages:

- Use very little memory (usually a constant amount)
- Find often reasonable solutions in large or infinite (continuous) state spaces.

Hill-climbing search

"Like climbing Everest in thick fog with amnesia"

- Moves in the direction of the highest value, expanding the best neighbour (aka *greedy local search*)
- Terminates when it reaches a peak (no neighbour with better value)
- Doesn't maintain a search tree, only the current node

Hill-climbing search

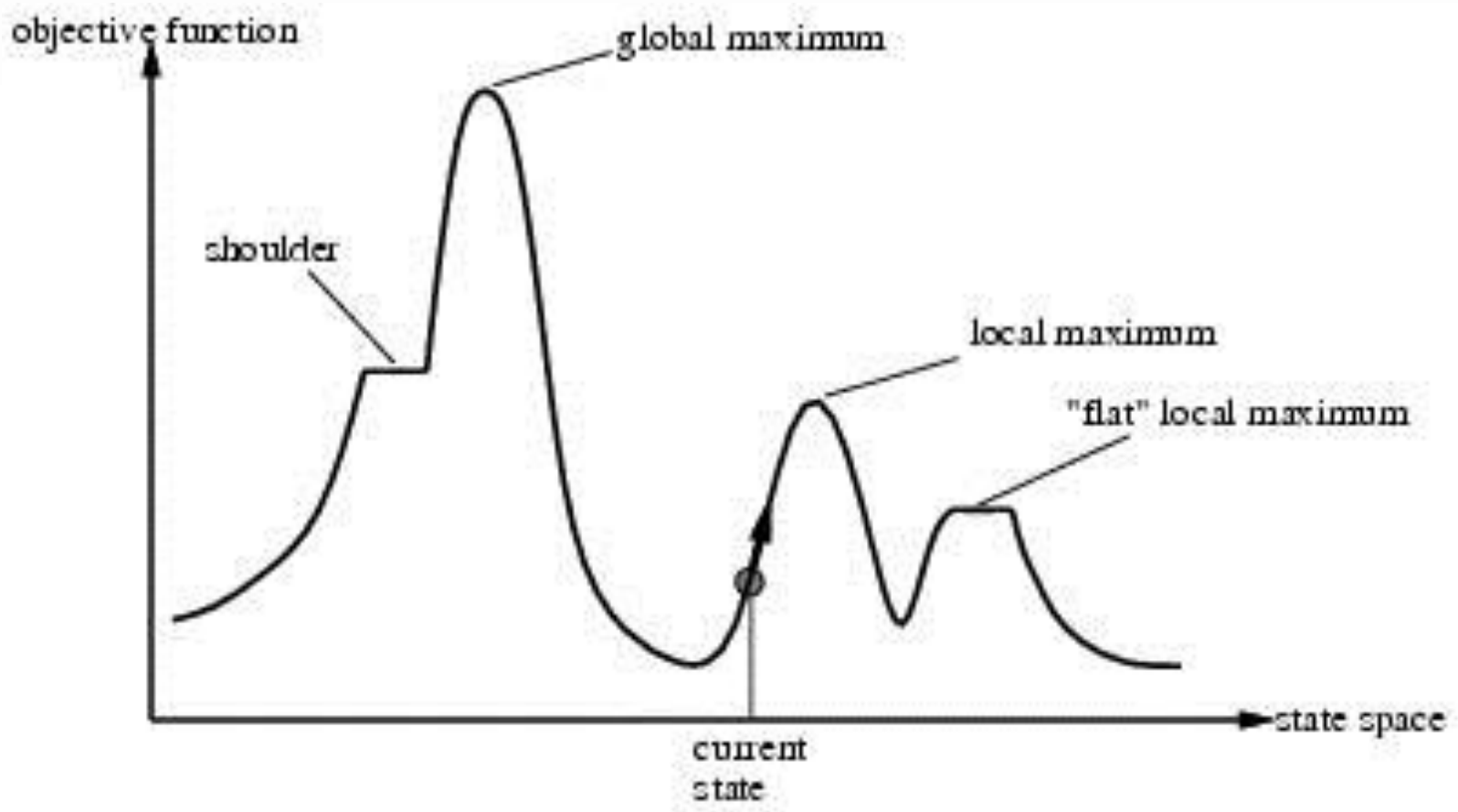
The algorithm does not maintain a search tree, so the current node data structure needs to only record the state and its objective function value.

Hill climbing does not look ahead beyond the immediate neighbors of the current state

> *"Like trying to climb mount Everest in a think fog while suffering from amnesia"*

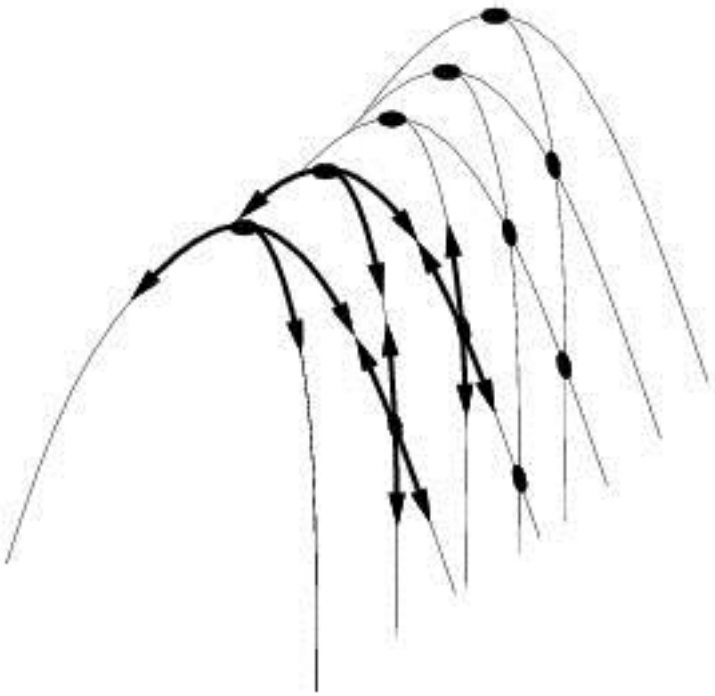
Hill-climbing a.k.a. *greedy local search*: because it grabs a good neighbor state without thinking ahead about where to go next. It turns out that greedy algorithms often perform quite well.

Drawbacks



Local maximum: a peak that is higher than each of its neighbors, but lower than the global maximum. HCS algorithms that reach the vicinity of a local maximum will be drawn towards the peak, but will then be stuck with nowhere else to go.

Drawbacks

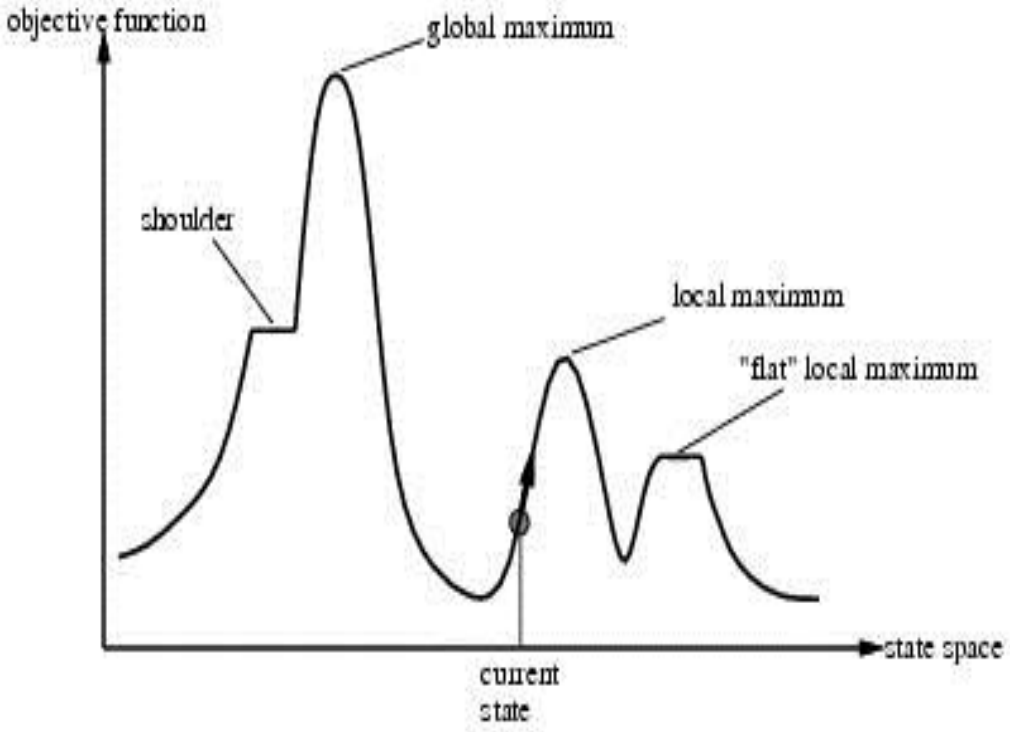


Ridge = sequence of local maxima difficult for greedy algorithms to navigate

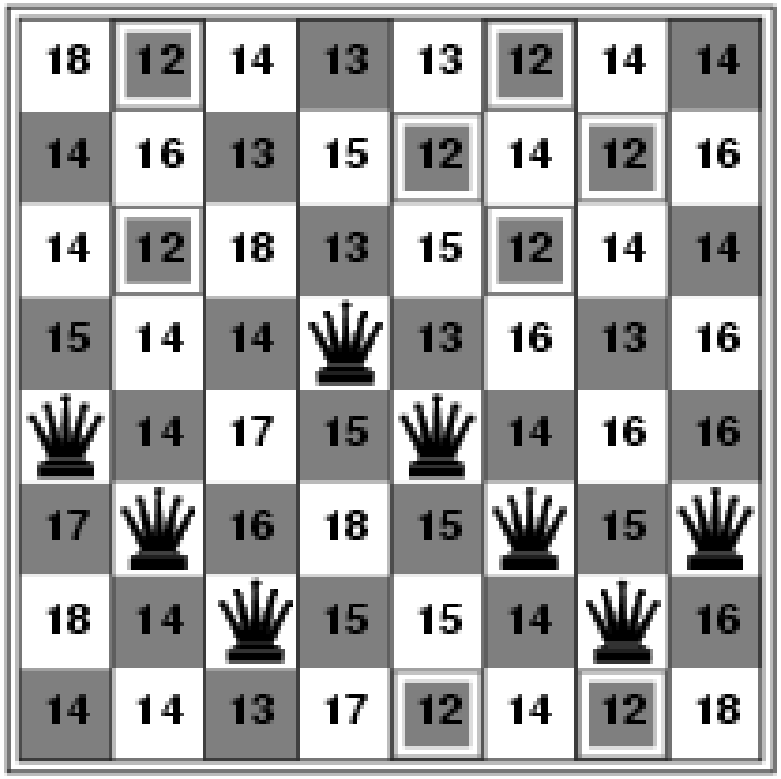
In the figure above a grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maxima all the available actions point downhill.

Drawbacks

Plateau = an area of the state space where the evaluation function is flat. It can be a flat local maximum from which no uphill exit exists or a shoulder, from which it is possible to make progress.



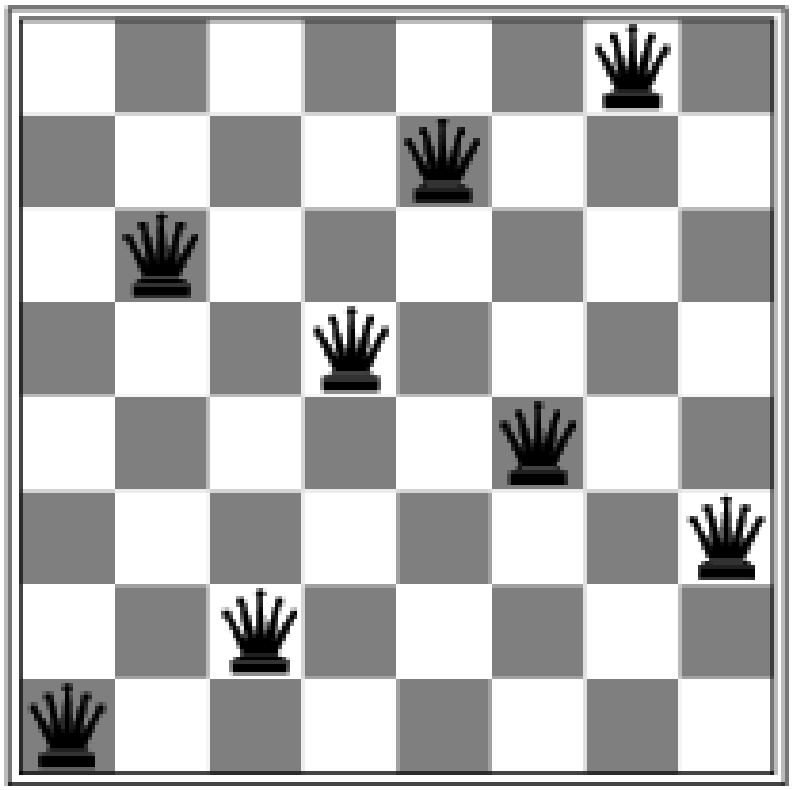
Hill-climbing search: 8-queens problem



h = number of pairs of queens that are attacking each other, either directly or indirectly

(h = 17 for the above state)

Hill-climbing search: 8-queens problem



- A local minimum with *h* = *1*

Simulated annealing

A hill-climbing algorithm that *never* makes downhill move towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.

In contrast, a purely random walk – that is moving to a successor chosen uniformly at random from the set of successors – is complete, but *extremely inefficient*.

A reasonable approach is to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness.

Simulated annealing is such an algorithm.

Simulated annealing

In metallurgy, annealing is the process used to temper and harden metals and glass by heating them to a high temperature and then gradually cooling them

To understand simulated annealing, let's switch our point of view from hill-climbing to the case where we are trying to minimize the cost. And imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.

- If we just let the ball roll, it will come to rest at a local minimum.
- The trick is to shake just hard enough to bounce the ball out of the local minima, but not hard enough to dislodge it from the global minimum.

The simulated annealing solution is to start by shaking hard (i.e. at a high temperature) and then gradually reduce the intensity of the shaking (i.e. lower the temperature).

Simulated annealing search

Simulated annealing search algorithm escapes local maxima by allowing some "bad" moves but gradually decreasing their frequency

Similar to hill-climbing but instead of picking the best move it picks a random move

- If the move improves the situation it is accepted
- If it doesn't improve it is accepted with some probability
  - The probability decreases with the "badness" of the move
  - The probability also decreases as the "temperature" goes down; bad moves are more likely at the start

It can be proven that if "temperature" is lowered slow enough the algorithm will find a global optimum with probability approaching 1.

Widely used in optimization tasks, scheduling, etc.

Part 2: Summary

Local Search
Hill-Climbing Search
Simulated Annealing

Local Beam Search

Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.

The local beam search algorithm keeps track of *k* states rather than just one:

1. It begins with *k* randomly generated states.
2. At each step, all the successors of all *k* states are generated.
3. If any one is a goal, the algorithm halts.
4. Otherwise it selects the *k* best successors from the complete list and repeats.

Local Beam Search

In a local beam search, useful information is passed among the *k* parallel search threads

At first sight, a local beam search with *k* states might seem to be nothing more than running *k* random restarts in parallel instead of in sequence.

In fact, the two algorithms are quite different.

- In a random-restart search, each search process runs independently of all the others.
- In a local beam search, if one state generates several good successors and the other *k-1* states all generate bad successors, then the effect is that the first state says to the others, "Come over here!"

The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made

Local beam search

Keep track of *k* states instead of one

- Initially: *k* random states
- Next: determine all successors of *k* states
- If any of successors is goal → finished
- Else select *k* best from successors and repeat.

Major difference with random-restart search

- Information is shared among *k* search threads.

Can suffer from lack of diversity.

- Stochastic variant: choose k successors at proportionally to state success.

Summary

Best-first search

- the minimum-cost unexpanded nodes are selected for expansion.
- uses heuristic *h(n)* (the cost of a solution from n)

Greedy best-first search

- expands nodes with minimal *h(n)*.
- not optimal but often efficient

A* search

- expands nodes with minimal *f(n) = g(n)+h(n)*
- complete and optimal, provided that *h(n)* is admissible
- high space complexity

Summary (cont.)

Heuristic functions

- The performance of a heuristic search depends on the quality of the heuristic function
- Good heuristic functions can be constructed by relaxing the problem or by constructing pattern databases

Local search methods (e.g. hill climbing)

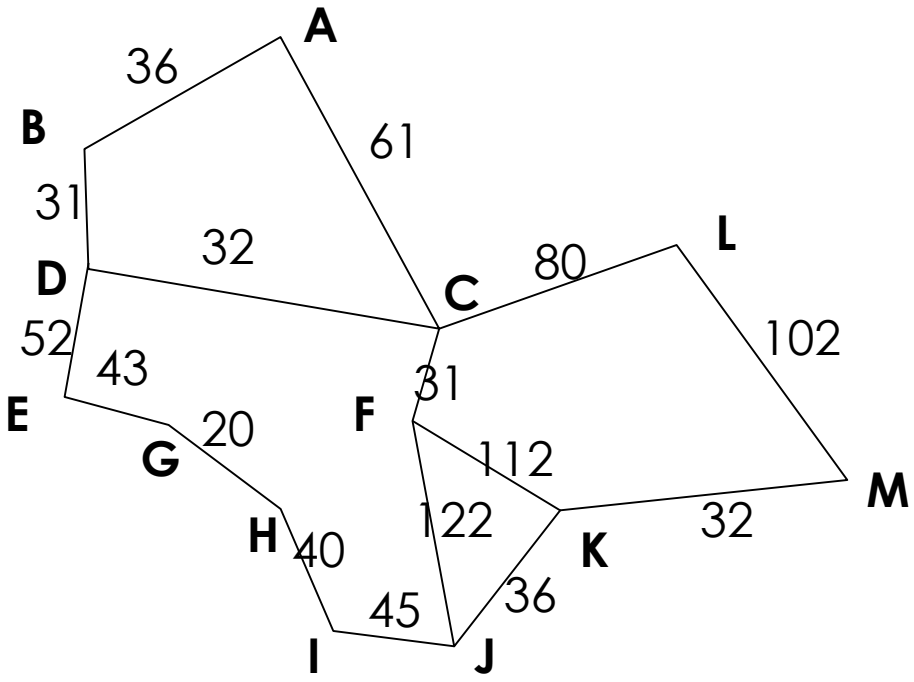
- do not keep the path
- operate on the complete-state formulations
- keep small number of nodes in the memory
- can stop in local maximum

Stochastic algorithms

- simulated annealing
- stochastic beam search

Exercise

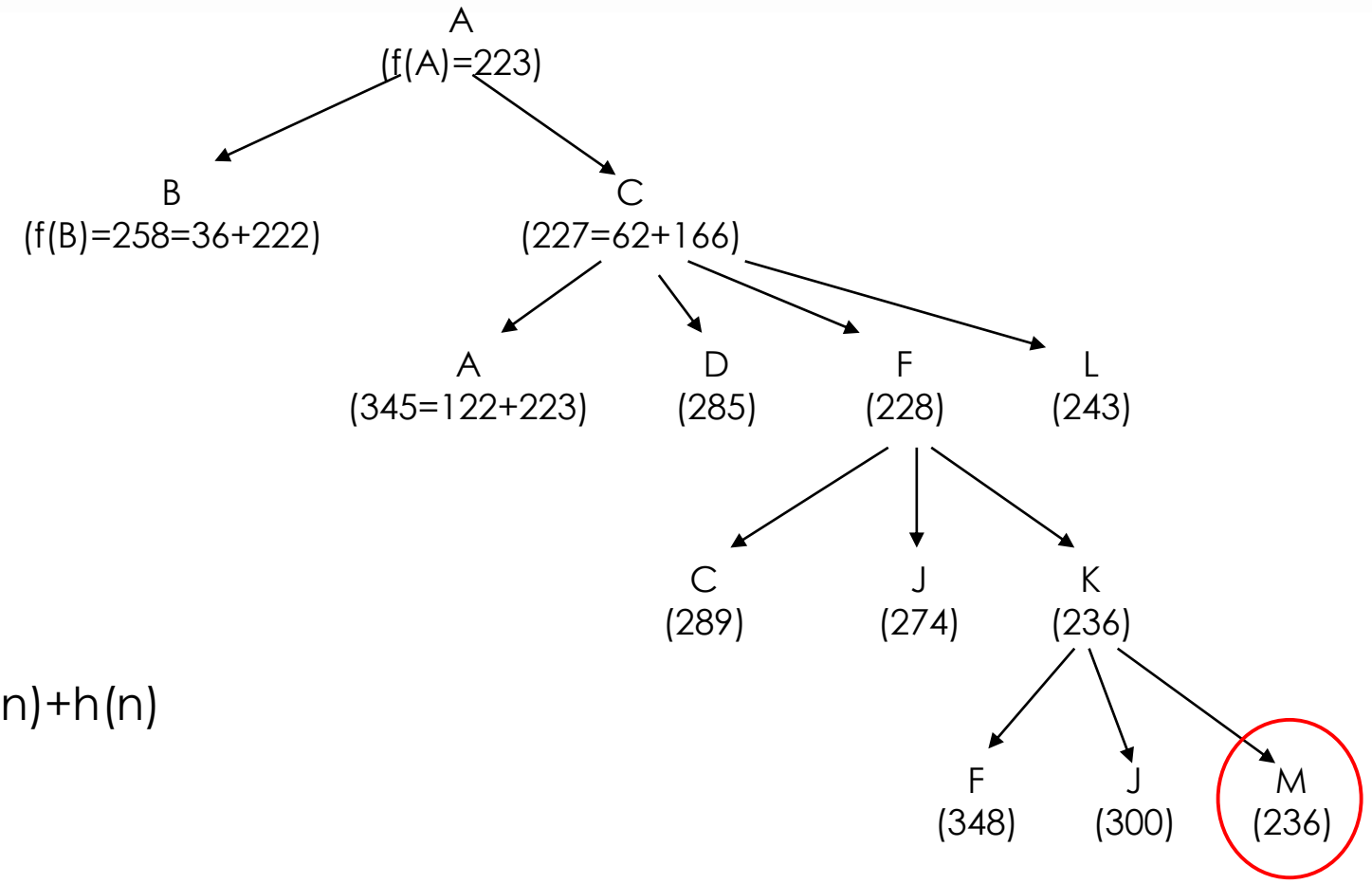
- Path from A to M using A*



Straight-line Distance to M

A	223
B	222
C	166
D	192
E	165
F	136
G	122
H	111
I	100
J	60
K	32
L	102

Solution



f(n) = g(n)+h(n)

Questions?